CHART2 NTCIP Detailed Design

Detailed Design, Development and Integration of the NTCIP Driver into the CHART2 System

Edwards and Kelcey Technical Design Document for CHART2.

This document contains the Detailed Design for the NTCIP Driver and procedures to integrate the driver into the CHART2 software system. The existing CHART2 software provides a series of interfaces for interaction with a new DMS type Driver. The driver developed by Edwards and Kelcey takes advantage of these interfaces to extend the communication protocols of CHART2 for the NTCIP standard. The NTCIP is a set of standards developed by several interested bodies for Intelligent Transportation Systems. This document describes the code architecture in detail for the NTCIP DMS Driver to be integrated with the CHART2 software.

Author	Cameron Riley & Nadia Trammell
Email	criley@ekmail.com & ntrammell@ekmail.com
Company	Edwards and Kelcey Technology, Inc
URL	
Address	750 Miller Drive, Suite F-1
City	Leesburg, Vi rginia
Zip	20175
phone	703.779.7988
fax	703.779.7989
date	May 24 , 2002

Purpose
This document describes the detailed design of the CHART2 NTCIP Driver and procedures to integrate the driver into the CHART2 software system.

Objectives
The main objective of this document is to provide software developers with details regarding the implementation of the NTCIP Driver. This document also serves to provide information to those outside of the software development community to show how the requirements are being accounted for in the software design.

Abstract

NTCIP is a grouping of standards for the meaningful communication between devices for ITS. The CHART2 system is the Maryland State Highway Administration's software system for monitoring and control of the ITS devices throughout the state for traffic management. This project is for the development and integration of an NTCIP driver for NTCIP compliant DMS's into the CHART2 software.

The CHART2 software provides strong interfaces for the integration of an NTCIP DMS into theCHART2 system. The central interface is the ProtocolHdlr interface which, determines thecommunication and interaction between the CHART2 DMS object and the physical message sign.

NTCIP defines the communication at the Data Link layer as PMPP or the Point to Multi Point Protocol. The operations for encoding and decoding the PMPP frames are separated into two packages. The NTCIPProtocolHdlr, when requested to communicate with the sign, uses the PMPP encoding package to create a PMPP frame and the decoding package when a response frame is received.

The PMPP protocol defines the information field as consisting of IPI and an SNMP PDU or alternatively an STMP PDU. The STMP protocol is not finalized and the SNMP is a more common Internet protocol. Edwards and Kelcey Technology evaluated three SNMP libraries, with joeSNMP being decided as the most suitable for CHART2's requirements.

The NTCIP standard defines the Mib groups for NTCIP compliance. The NTCIP driver uses MibXML to store the MIB values locally in the CHART2 system. The MibXML is DMS instance specific. When requests from the CHART2 system are made against the NTCIPProtocolHdlr interface, the status of the NTCIPDMS will be maintained by manipulation of the MibXML in the instance of the CHART2 NTCIPDMS.

The CHART2 DataPort interface is the session manager for opening direct communication with a Sign. The interface is a master only and doesn't support unsolicited asynchronous "trap" requests.

CHART

CHART is the highway management program of the Maryland State Highway Administration. CHART is comprised of four major components, traffic monitoring, incident response, traveler information and traffic management. The traveler information component of CHART provides real-time information concerning travel conditions on main roads in the primary coverage area. The information is conveyed to travelers via either Dynamic Message Signs (DMS), Highway Advisory Radio (HAR), commercial radio and television broadcasts and a telephone advisory service. The dynamic message signs are programmable message boards, both permanent and portable which are capable of displaying real-time messages such as traffic conditions to motorists. NTCIP compliant Dynamic Message Signs are a type of DMS that utilize the standard protocol as the means of communication. CHART2 is the current implementation of the CHART system being developed by the Maryland State Highways Administration.

CHART2 CORBA Object Transport Mechanism

The CHART2 System uses CORBA, to transport and communicate between objects across the distributed client and server system. The GUI's from which the DMS devices are managed, exist as clients in the CORBA system. The FMSServer carries the implementations of the objects. The driver interacts across this system by the IDL interfaces.

The ORB is a message bus between objects that may reside on any machine in the network. The IDL is the interfaces by which the distributed objects publish their capabilities. CORBA is the specification that describes the ORB's functionality. CORBA specifies location transparency and language transparency. For CHART2's distributed system, the former is the most important.

NTCIP Standard

The National Transportation Communications for Intelligent Transportation System Protocol (NTCIP) is a family of standards maintained by NEMA, AASHTO and ITE. The NTCIP standards provide the rules and vocabulary for electronic traffic equipment from different manufacturers to communicate and operate with each other. NTCIP compliant devices must follow this standard. For a Device to be NTCIP compliant it must implement a mandatory set of MIBs, accept data transport by PMPP and be capable of reading Multi Message Format for sign display.

Overview of Components

The major components of the NTCIP Driver for the CHART2 system are the CHART2 DMS object, the MibXML state storage mechanism, the Protocol Handler, the PMPP component and the DataPort.

The DMS object is the internal memory representation of a sign in the CHART2 software system. The DMS knows how to describe itself and maintains it's internal state and status. The DMS can use the Protocol Handler to communicate with the physical sign and query, or update its status including the message. It also stores the information that is displayed to the user through the GUI interface.

The MibXML component creates the Mib points and groups placed into a DMS instance specific XML structure in memory. The MibXML maintains the DMS's internal state including updating as the software communicates with the physical sign.

The NTCIP Protocol Handler is an extension to the existing DMSProtocol handler. for managing the actions specific to a NTCIP DMS. As described later, the Protocol Handler is the intermediary between the CHART system and the physical DMS, by providing the DataPort, which provides the logic for connecting to a physical DMS. The DataPort is the CHART2 interface for connecting across a physical line to a physical sign.

The PMPP component is the means for transportation of data meaningfully between the CHART2software system and the physical sign's onboard controller, including the Management Information Base. The PMPP protocol follows the NTCIP set of standards for communication with field devices.

CHART2.DMSProtocol Package

The CHART2 system allows for the software to be divided into small packages, which focus on particular tasks. The central component of the NTCIP driver is the Protocol Handler. This is represented for message signs through the *DMSProtocolHdlr* Interface. The NTCIP specific protocol handler implements this interface. The *NTCIPProtocolHdlr* has the following methods.

```
public void setConfiguration(DMSProtocolHdlrConfig config)
        throws DMSProtocolHandlerException;

public DMSProtocolHdlrConfig getConfiguration();

public void setMessage(DataPort port,
                       String multiMsg,
                       boolean beaconState)
        throws DMSProtocolHandlerException;

public void blank(DataPort port)
        throws DMSProtocolHandlerException;

public DMSDeviceStatus getStatus(DataPort port)
        throws DMSProtocolHandlerException;

public void reset(DataPort port)
        throws DMSProtocolHandlerException;

private void set(OID oid, DataPort dataport)
         throws UnsupportedEncodingException,
                InvalidFrameException,
                DataPortIOException,
                PmppErrorException

private String get(OID oid, DataPort dataport)
         throws UnsupportedEncodingException,
                InvalidFrameException,
                DataPortIOException,
                PmppErrorException

private byte[] activateMessage(int iMemoryType,
                               int iMessageNumber,
                               String crcValue)
```

The setConfiguration() method sets the configuration parameters, which are mostly used for determining formatting of messages. The getConfiguration() method is an accessor method for returning the configuration object. The reset() method resets the Sign's controller by setting the dmsSWReset mib value to 1. The set() method is used to access the data port and sending the request to the DMS sign. The get() method is also used to access the data port and retrieve a value from the DMS sign.

The blank() method blanks the sign removing any currently displayed message. Blanking a sign involves activating a message using memory type (7) blank. It should be noted that the CRC value for a blank message is always ‘0’.

The getStatus() method returns the current status of the sign as a CHART2 DMS status object. The getStatus method initializes the NTCIPDMSDeviceStatus object and sets the status values to those values returned from the sign. The following are the Mib values that are collected from the sign: control mode, source mode, message owner, power source, fuel level, volts of sign, engine RPM, beacon value, current message, and short error status.

The setMessage() method is for displaying a message on a Sign. Several steps need to be implemented in order to successfully display a message. When setting a message, Memory type (3), changeable is used. The first step is to set the Message Status to (6), which is Request Modify. Once the Message Status returns a (2), Modifying, the sign is ready to receive a message. The beacon status, message owner, run time priority, and multi-message string are then set in the sign with the desired values. The message status is then set to (7) Request Validate. A value of (2), Validated, is retrieved from the sign to complete the task of sending the message to the sign. The final step in the setMessage method is the activation of the message on the sign. To activate a message set the dmsActivateMessage mib to the MessageActivationCode string. The MessageActivationCode is constructed with the createMessageActivationCode() method.

The createMessageActivationCode() method returns the activate message code as a byte array under Octet String BER encoding rules. The activate message code is used to activate a message. The mib dmsActivateMessage uses the syntax MessageActivationCode to activate a message. The dmsActivateMessage is set with a code indicating the message, which the sign shall activate. The NTCIP 1203 Amendment 1 states the following about MessageActivationCode:

> MessageActivationCode = OCTET STRING (SIZE(12))
> The MessageActivationCode consists of those parameters required to activate a message on a DMS. It is defined as an OCTET STRING containing the BER-encoding of the following ASN.1 structure.
>
> The MessageActivationCodeSturucture = SEQUENCE

Duration	16 bits
ActivatePriority	8 bits
MsgMemoryType	8 bits
MessageNumber	16 bits
MessageCRC	16 bits
SourceAddress	32 bits

In order to activate a message you must set the dmsActivateMessage mib with the MessageActivationCode value. This is accomplished by constructing an octet string consisting of duration, activate priority, message memory type, message number, message CRC and source address. This octet string represents the MessageActivationCode that the dmsActivateMessage is set with.

For example by using the following values:

duration -1 -1
priority -128
memory 3
number 1
crc -87
ip address 10 2 34 54

When activating a message calling the method createMessageActivationCode returns the MessageActivationCode as a byte array under Octet String BER encoding rules. See below.

```
/*
 * Returns the Message Activation Code as an byte array
 * under Octet string BER encoding rules
 * @param iMemoryType, the Memory Type of the message
 * @param iMessageNumber, the The Message Number
 * @param crcValue, the CRC Value of the message
 */
public byte[] createMessageActivationCode(int iMemoryType,
                          int iMessageNumber,
                          String crcValue)
{


    /* Create the MessageActivationCode string to set
     * the value of "dmsActivateMessage"
     * Consist of
     * duration = 2 bytes
     * priority = 1 byte
     * IDCode = 5 bytes (inclueds memory type = 1 byte
     *                       MsgNumber = 2 bytes
     *                       CRC= 2 bytes)
     * IP Address = 4 bytes off of machine its running on
     */
    PrimitiveUtility pu = new PrimitiveUtility();


    /*
     * Duration as two bytes
     */
    int duration = 65535;
    byte durByte1 = (byte)(duration & 0xff);
    byte durByte2 = (byte)((duration >> 8) & 0xff);

    /*
     * Must be greater than runtime priority
     * If this value is greater than or
     * equal to the dmsMsgRunTimePriority
     * of the current message, the
     * new message will be displayed
     */
     int priority = 255;
     byte priorityByte = (byte)(priority & 0xff);


    /*
     * int IDCode = iMemoryType + iMessageNumber + crc;
     */

    /*
     * MemoryType as a byte
     */
    byte memoryByte = (byte)(iMemoryType & 0xff);

    /*
     * Message Number as a byte
     */
    byte msgNumByte1 = (byte)((iMessageNumber >> 8) & 0xff);
    byte msgNumByte2 = (byte)(iMessageNumber & 0xff);

    /*
     * CRC as a byte
     */
    int crcv = Integer.parseInt(crcValue);
    byte crcByte1 = (byte)((crcv >> 8) & 0xff);
    byte crcByte2 = (byte)(crcv & 0xff);

    /*
     * Get the ipaddress of the machine
     * and return 4 bytes representing
```

```
     * the ipaddress
     */

    ipinfo = new IPInfo();
    String ipa = ipinfo.getAddressAsString();

    byte ipByte1 = ipinfo.getFirstByte(ipa);
    byte ipByte2 = ipinfo.getSecondByte(ipa);
    byte ipByte3 = ipinfo.getThirdByte(ipa);
    byte ipByte4 = ipinfo.getFourthByte(ipa);

    /*
     * Message Activation Code as Byte
     */
          byte[] MAC = new byte[]{durByte1, durByte2, priorityByte, memoryByte, msgNumByte1, msgNumByte2,
        crcByte1, crcByte2, ipByte1, ipByte2, ipByte3, ipByte4};

    return MAC;

  }
```

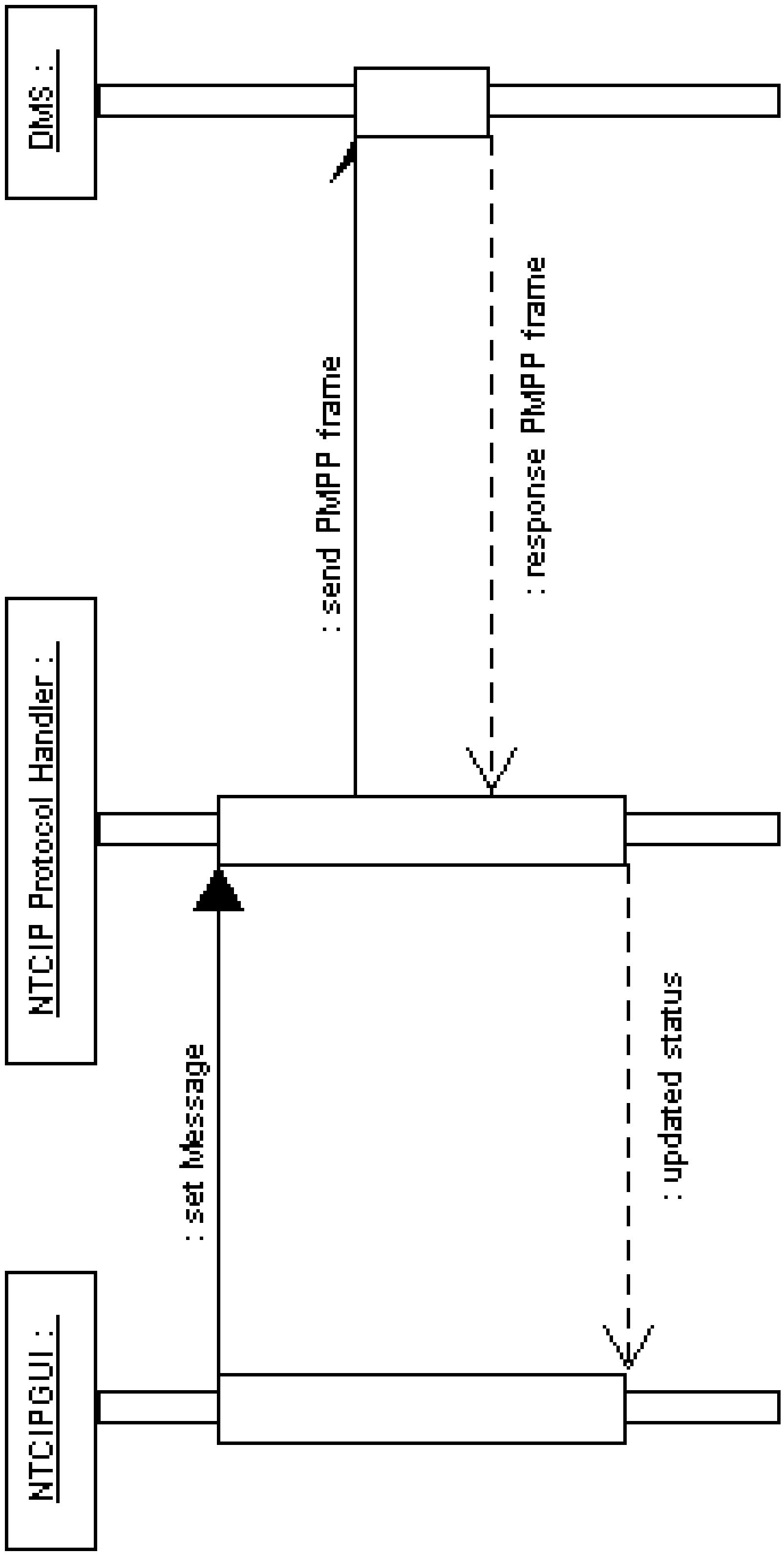
The createMessageActivationCode method would return the following MessageActivationCode:

Byte [] = [FF FF 80 03 00 01 11 A9 0A 02 22 36]

The CHART2 system makes requests on these methods and the Protocol Handler creates suitable PMPP requests, which update the DMS's current state, which is then reflected back into the CHART2 system.

Also included in this package is NTCIPDMSDeviceStatus and NTCIPErrorFactoryClass. NTCIPDMSDeviceStatus extends the DMSDeviceStatus class object. The NTCIPDMSDeviceStatus class contains status data specific to the NTCIP DMS. This status data includes the following mib values: control mode, source mode, message owner, power source, fuel level, volts of sign, engine RPM, beacon value, current message, and short error status. There is only one method within the NTCIPDMSDeviceStatus class, which returns all of the status data collected as a string.

The NTCIPErrorFactoryClass is responsible for handling mib object errors. If an error occurs while validating or activating a message, the mibs dmsValidateMessagError and dmsActivateMessageError will return a value, which will indicate the reason for the occurrence of the error. If the value indicates that it is a multi-syntax error the dmsMultiSyntaxError mib object will return a value for what type of multi-syntax error occurred. The name of the mib object and the number value are used to initialize the NTCIPErrorFactoryClass. The NTCIPErrorFactoryClass has a method for each error mib object, which includes: dmsValidateMessageError, dmsActivateMessageError and dmsMultiSyntaxError. These methods are responsible for identifying the number value of that particular mib object and returning a full description of the value in the form of a string.



Sequence Diagram describing the interaction between the GUI, Protocol Handler and the Sign

CHART2.Utility Package

The manipulation of the PMPP packet is done through byte arrays in the software. For this reason, the package requires a set of methods to manipulate byte arrays. The *PrimitiveUtility* class is used to get around the problems java has with unsigned data types.

The *FileUtility* class is used for common manipulations on File's. The *FileUtility* class only exposes one method, getFile(). This method is used to create an input stream from a file object.

The *IPInfo* class is a Utility object with methods for collecting IP address Information. The *IPInfo* has the following methods.

```
public java.lang.String getAddressAsString()

public byte[] getAddress()

public byte getFirstByte(java.lang.String b)

public byte getSecondByte(java.lang.String b)

public byte getThirdByte(java.lang.String b)

public byte getFourthByte(java.lang.String b)
```

The getFirstByte() method removes the "." from the IP address and returns the first section as a byte. The remaining methods getSecondByte(), getThirdByte, and getFourthByte return the IP section it refers to.

The *XMLUtility* class is used for common manipulation on XML files and objects. The methods include:

```
public Document getXml(File file)
        throws IOException, JDOMException

public Document getXml(InputStream inputstream)
        throws IOException, JDOMException

public String getName(Document document)

public String getName(InputStream inputstream)
        throws IOException, JDOMException
```

CHART2.Utility.Communication Package

The Protocol Handler interface and the CHART2 system expose the DataPort as the communication session interface. The *DataPort* IDL interface supports the following methods;

```
public void send(byte[] data)
throws DataPortIOException;

public byte[] receive(long initialTimeoutMillis,
        long interCharTimeoutMillis,
        long maxReadDurationMillis)
        raises (DataPortIOException);
```

Where the send() method sends the binary over the port and the receive() method receives bursts of bytes from the port as the data chunks become available. The time the port remains listening can be specified through the initialTimeoutMillis which is the length of time to wait until the first byte of data is received, the interCharTimeoutMillis which is the amount of time to wait between two consecutive bytes. Once this times out it is assumed that the complete packet has been received. The final argument is the maxReadDurationMillis, which is the maximum amount of time that should be spent receiving bytes after the initial byte is received.

The PMPP standard defines Traps for SNMP. The DataPort is a master that opens a port and sends binary data and then waits to receive a reply. Traps are asynchronous, unsolicited calls from the Sign. The trapping mechanism requires that the system be capable of fielding unsolicited PMPP calls from the Sign to the CHART2 system. The DataPort structure doesn't support this type of communication.

As SNMP is a widely used and adopted communication standard there are several SNMP libraries available for use, both commercial and open source. Edwards and Kelcey Technology used joeSNMP because it had the richest API and had the to byte array functions which allowed an easier integration into CHART2.

The PMPP package is separate from the Protocol Handler and placed in the CHART2.Utility.Communications package. The package will have an object, which follows the Façade pattern and provides a simple enclosing interface to the PMPP package. It is through this façade object that the CHART2 system should access the packages capabilities. The facade object, *PmppBuilder*, carries four methods;

```
public byte[] encodeGet(byte[] address, OID oid)

public byte[] encodeSet(byte[] address, OID oid, String value)

public String decodeGet(byte[] buffer)
        throws InvalidFrameException

public void decodeSet(byte[] buffer)
        throws InvalidFrameException
```

Where the encodeGet() method creates a binary PMPP frame to get the value of the OID, the encodeSet() method creates a binary PMPP frame to set the value of the OID in the sign, the decodeGet() method unmarshalls the value from the received binary chunk and the decodeSet() method checks the frame from the received binary chunk. The address binary comes from the CHART2DMS's getId() method. Through these four methods the requirements for interaction with the NTCIP compliant DMS as defined in the Protocol Handler interface are achieved.

The working object in the communications package is the PmppRequest object. This object is tasked with storing the frames state, determining it's validity to the standard, as well as marshalling and unmarshalling the frame to an HDLC encoded frame and decoding the frame to it's constituent parts. The *PmppRequest* follows the *PmppSyntax* interface, which contains the methods;

```
public int encodePmpp(byte[] buf,
                      int offset,
                      HdlcEncoder encoder);

public int decodePmpp(byte[] buf,
                      int offset,
                      HdlcEncoder encoder)
        throws InvalidFrameException;
```

The method encodePmpp() accepts a binary array, and encodes the bytes contained within to an HDLC encoding from the offset onwards. The returned int is the last encoded byte in the buffer. The decodePmpp() method decodes the HDLC in the buffer from the offset onwards. The offset represents the beginning of Pmpp frame. The returned int is the offset of the last decoded byte. The decodePmpp method throws an InvalidFrameException if the buffer fails to be decoded due to frame errors. The HdlcEncoder object is analogous to the AsnEnoder and BerEncoder objects found in the joeSNMP library.

The HdlcEncoder object is aware of the requirements for transparency that the HDLC standard places on PMPP frames. Transparency requires that the 0x7E's and 0x7D's in the frame be escaped with 0x7D 0x5E and 0x7D 0x5D respectively. The frame flags which are 0x7E are not included in the transparency. The HdlcEncoder class escapes and unescapes 0x7E's and 0x7D's from a buffer. The *HdlcEncoder* class has the methods:

```
public int escape(byte buffer[],
           int offset,
           int length)

public byte[] escapeByte(byte buffer[],
               byte escape[],
               int offset)

public byte[] unescapeByte(byte buffer[],
                byte b,
                int offset,
                int length)

public byte[] unescape(byte buffer[],
             int offset,
             int length) throws CHART2.Utility.Communications.InvalidFrameException
```

The PMPP standard requires that a CRC-16 CCITT checksum be calculated and embedded into the FCS field. The FCS is the checksum on the frame to determine if there are transmission errors. The FCS check during encoding is done after the SNMP Request, Address and Control fields have been added to the PMPP. For unmarshalling this is done after the PMPP packet is de-framed. The FCS check is called from the PmppRequests encodePmpp() and decodePmpp() methods. The working object for the CRC check is the CRC16 object. This object follows the java.util.zip.Checksum interface. The standard Sun java libraries include a CRC32 object, but no published CRC16 object. There is in the sun.misc package a CRC16 object, but it is undocumented and only appears with Sun JVM's. It also does not follow the Checksum interface. The Checksum interface has the methods;

```
public long getValue();

public void reset();

public update(int b);

public void update(byte[] b, int offset, int length);
```

Where the checksum is calculated through the addition of bytes by the update methods and getValue() is the value of the checksum's value, which can be obtained at any time. The concrete class for *CRC16* adds the two publicly available methods;

```
public int getHighByte()

public int getLowByte()
```

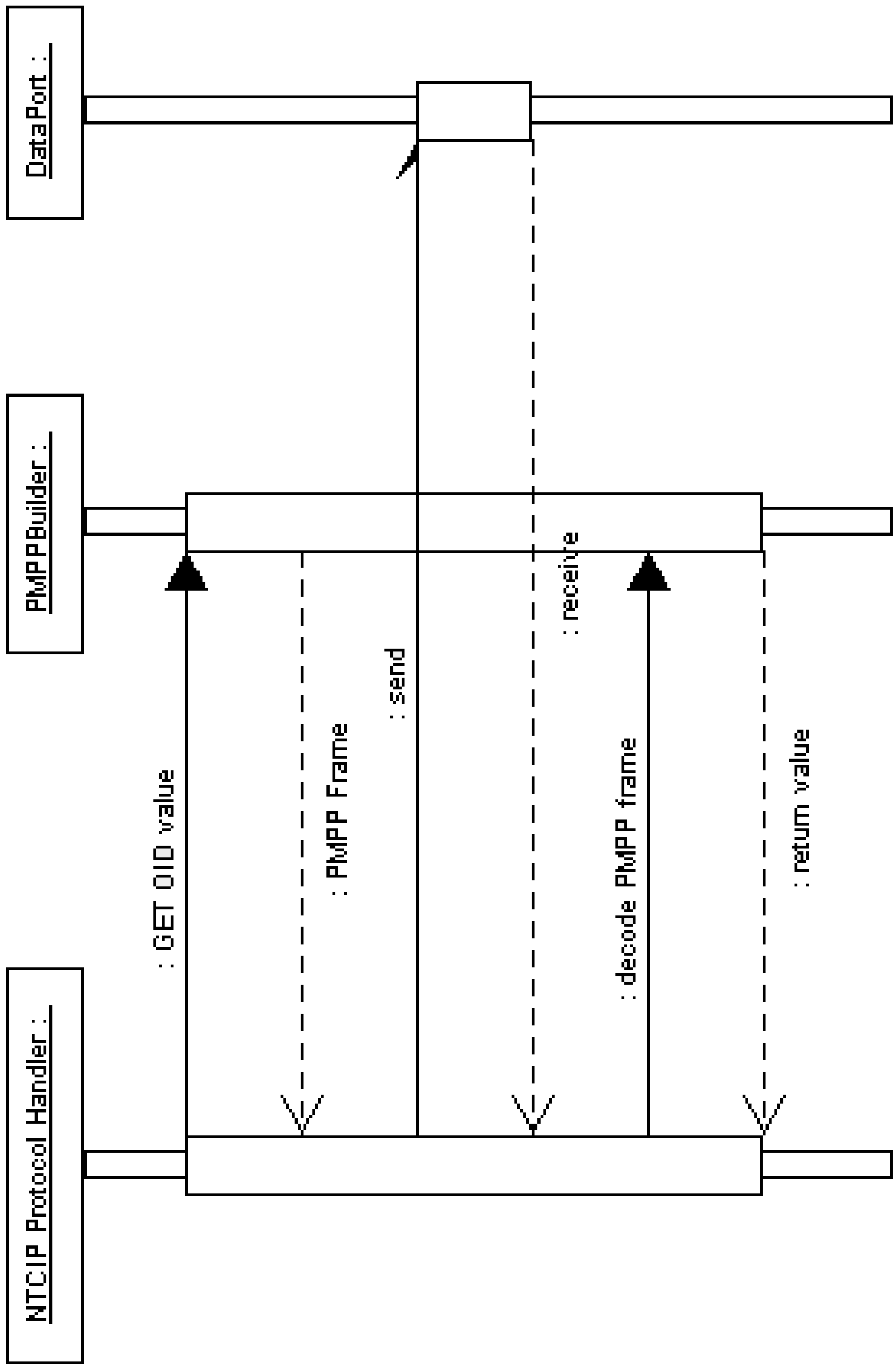
Which facilitates the checksum value being added quickly to the *PmppRequest* as the 16 bit field the PMPP standard requires.

The *SnmpPdu* object is the CHART2 facade class to the joeSNMP library. The *SnmpPdu* object accepts through its constructors the standard GET, SET and RESPONSE requests. The constructors are;

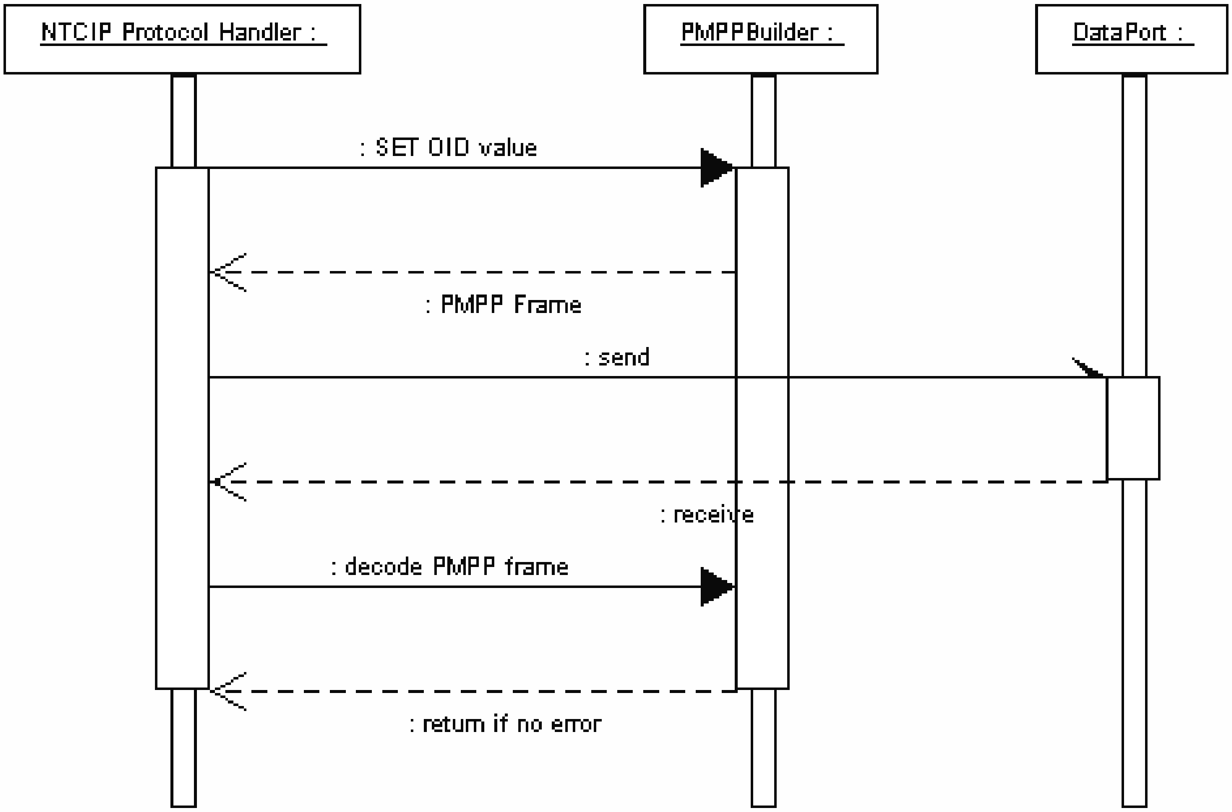
```
public SnmpPdu(OID oid, String value)
        throws UnsupportedEncodingException

public SnmpPdu(OID oid)
        throws UnsupportedEncodingException
```

```
public SnmpPdu(byte[] data)
        throws UnsupportedEncodingException
```

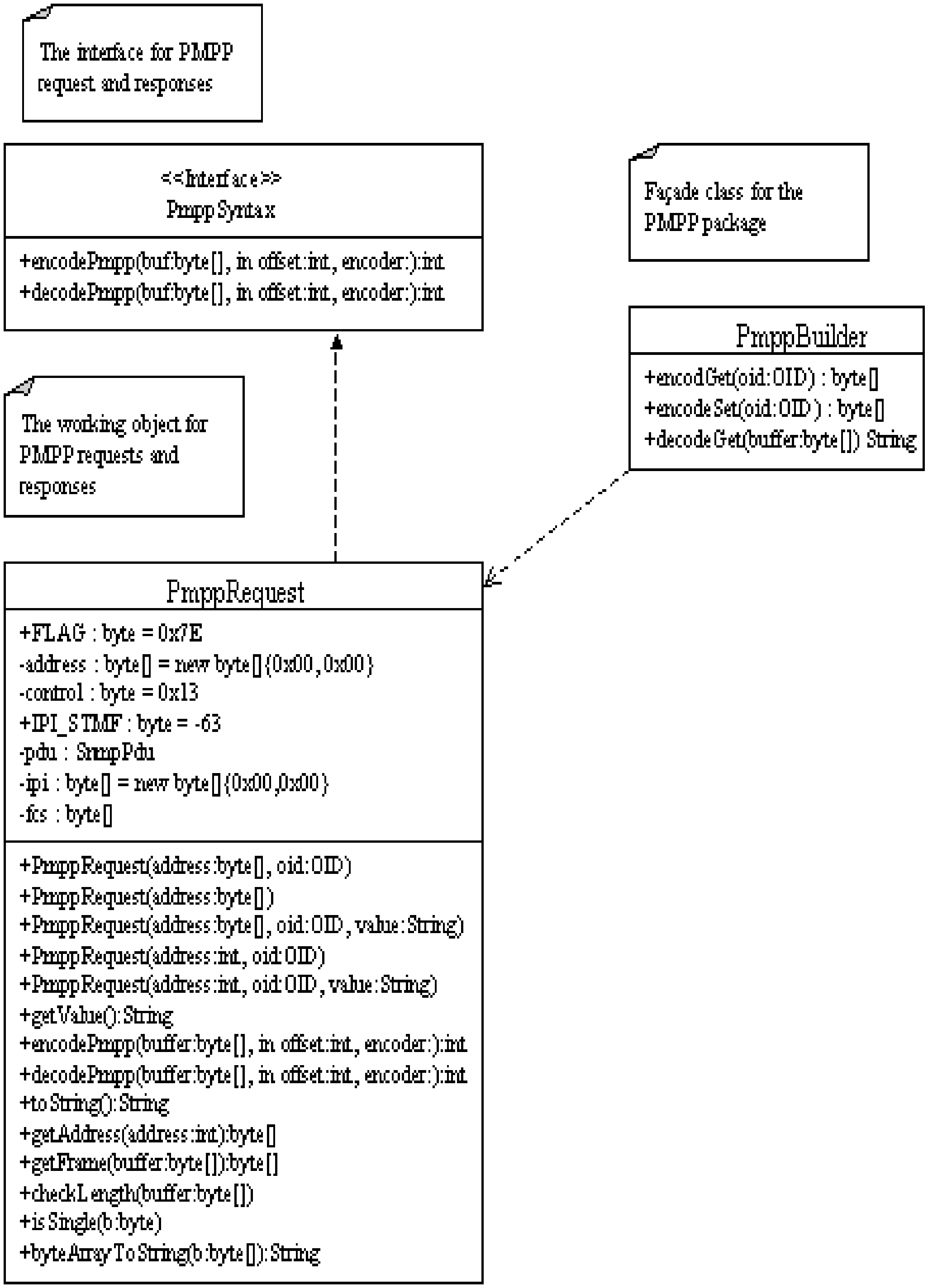
The constructor which accepts an OID and value as argument is the constructor for an SNMP SET request the constructor which accepts an OID is an SNMP GET request and the constructor, which accepts a binary array as an argument, is the SNMP RESPONSE constructor. The SnmpPdu also contains encodeAsn() and decodeAsn() methods, which are for marshalling and unmarshalling SNMP requests to and from BER Encoding. The remaining methods in the SnmpPdu are getErrors(), getLength() and getValue().



Sequence Diagram for a GET request describing the interaction between the NTCIP Protocol Handler, the PmppBuilder and the DataPort.



Sequence Diagram for a SET request describing the interaction between the NTCIP Protocol Handler, the PmppBuilder and the DataPort.


The interface for PMPP request and responses
<<Interface>>
PmppSyntax
+encodePmpp(buf:byte[], in offset:int, encoder:):int
+decodePmpp(buf:byte[], in offset:int, encoder:):int
Façade class for the PMPP package
PmppBuilder
+encodGet(oid:OID) : byte[]
+encodeSet(oid:OID) : byte[]
+decodeGet(buffer:byte[]) String
The working object for PMPP requests and responses
PmppRequest
+FLAG : byte = 0x7E
-address : byte[] = new byte[]{0x00,0x00}
-control : byte = 0x13
+IPI_STMF : byte = -63
-pdu : SnmpPdu
-ipi : byte[] = new byte[]{0x00,0x00}
-fcs : byte[]
+PmppRequest(address:byte[], oid:OID)
+PmppRequest(address:byte[])
+PmppRequest(address:byte[], oid:OID, value:String)
+PmppRequest(address:int, oid:OID)
+PmppRequest(address:int, oid:OID, value:String)
+getValue():String
+encodePmpp(buffer:byte[], in offset:int, encoder:):int
+decodePmpp(buffer:byte[], in offset:int, encoder:):int
+toString():String
+getAddress(address:int):byte[]
+getFrame(buffer:byte[]):byte[]
+checkLength(buffer:byte[])
+isSingle(b:byte)
+byteArrayToString(b:byte[]):String

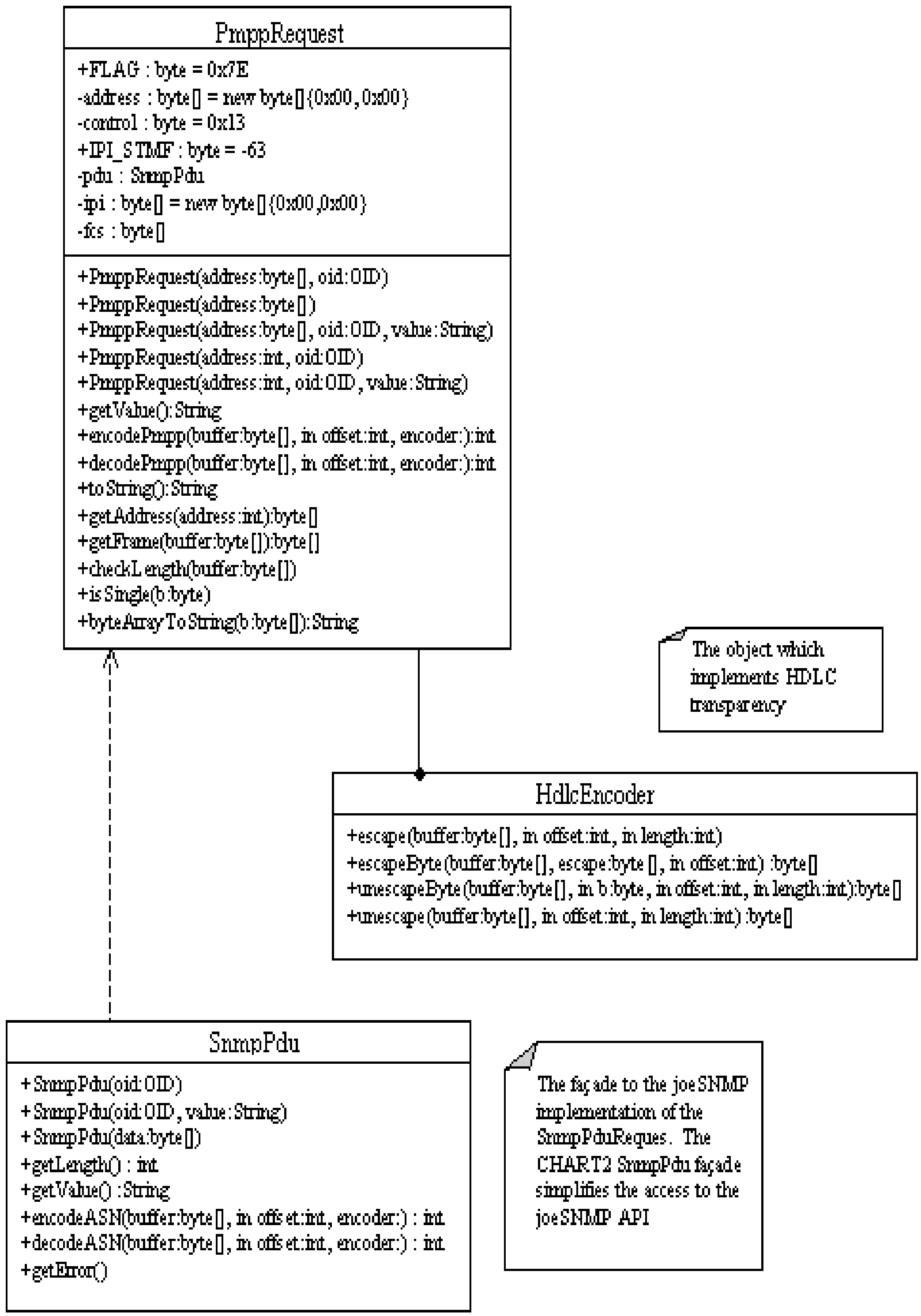

PmppRequest
+FLAG : byte = 0x7E
-address : byte[] = new byte[]{0x00,0x00}
-control : byte = 0x13
+IPI_STMF : byte = -63
-pdu : SnmpPdu
-ipi : byte[] = new byte[]{0x00,0x00}
-fcs : byte[]
+PmppRequest(address:byte[], oid:OID)
+PmppRequest(address:byte[])
+PmppRequest(address:byte[], oid:OID, value:String)
+PmppRequest(address:int, oid:OID)
+PmppRequest(address:int, oid:OID, value:String)
+getValue():String
+encodePmpp(buffer:byte[], in offset:int, encoder:):int
+decodePmpp(buffer:byte[], in offset:int, encoder:):int
+toString():String
+getAddress(address:int):byte[]
+getFrame(buffer:byte[]):byte[]
+checkLength(buffer:byte[])
+isSingle(b:byte)
+byteArrayToString(b:byte[]):String
The object which implements HDLC transparency
HdlcEncoder
+escape(buffer:byte[], in offset:int, in length:int)
+escapeByte(buffer:byte[], escape:byte[], in offset:int) :byte[]
+unescapeByte(buffer:byte[], in b:byte, in offset:int, in length:int):byte[]
+unescape(buffer:byte[], in offset:int, in length:int) :byte[]
SnmpPdu
+SnmpPdu(oid:OID)
+SnmpPdu(oid:OID, value:String)
+SnmpPdu(data:byte[])
+getLength() : int
+getValue() :String
+encodeASN(buffer:byte[], in offset:int, encoder:) : int
+decodeASN(buffer:byte[], in offset:int, encoder:) : int
+getError()
The façade to the joeSNMP implementation of the SnmpPduReques. The CHART2 SnmpPdu façade simplifies the access to the joeSNMP API

CHART2.Utility.Mib Package

When the Protocol Handler acts upon requests from the operator at the CHART2 GUI, the Protocol Handler makes requests with the PMPP Protocol to the Sign and receives responses, which reflects the DMS's current state. The Sign's state is stored in the actual device as a MIB or Management Information Base. A highly nodular standard, which contains MIB points that store specific information about the device. These MIB points are also referred to as objects. Each MIB point contains information on the ID, name, value, type of value, and read/write permission. MIBs follow the ASN.1 standard, which is commonly stored in a flat file format. The NTCIP set of standards supports the MIB series, TS-3.2, TS-3.3, TS-3.5, TS-3.6 and TS-3.7, of which some groups are mandatory and others are optional. Not all the MIBs are required to support the Protocol Handler interface. The mandatory MIB groups include;

- Configuration Conformance Group
- Security Node Conformance Group
- Sign Configuration and Capability Conformance Group
- Font Definition Conformance Group
- DMS Configuration Conformance Group
- Multiconfiguration Conformance Group
- Message Table Conformance Group
- Sign Control Conformance Group
- Illumination/Brightness Conformance Group
- Scheduling Conformance Group
- Sign Status Conformance Group

To maintain the status of the DMS stored in the CHART2 system memory, while maintaining separation of the Model(Mib), View(DMSGUI) and Controller(DMS), it would be best stored in the DMS Object as an instance specific Mib database using XML as the structure. This allows for the knowledge of the MIBs to be segregated from the Java code describing the DMS and the Protocol Handler. Other benefits include the ASN.1 nodular structure being stored in a well supported data structure with many libraries to store, search and manipulate the nodes, coarser information of the Mib point such as OID, name and description. The original Mib structure can be loaded once and cloned for instance specific DMS states.

The Mib objects are contained in the CHART2.Utility.Mib package, which exists in the CHART2 system to create, copy, store, request, insert and manipulate MIBs providing an accessible and simple interface to manage a CHART2 DMS's current state.

The facade object for the Mib package is the *MibService* object. *MibService* contains methods for accessing the MIbdb through the *MibManager* interface. The *MibService* exposes three publicly available static methods;

```
public static List getGroupNames(MibManager mibmg)

public static OID getOID(MibManager mibmg,
                                        String type)
        throws OIDNotFoundException, OIDNotValidException,
                JaxenException, SAXPathException

public static void setOIDValue(MibManager mibmg,
                                                OID oid,
                                                String value)
```

The method getGroupNames() gets a listing of a Group's objects from the *MibManager* interface of the group that contains the specified MIB point. The method, getOID() returns an OID object from the object adhering to the *MibManager* interface and with the mib-name or mib-type by

ASN.1 parlance. The method setOIDValue() sets the value of the OID. The *MibManager* interface contains methods to interact with a Mib database. The interface follows;

```
public static List getGroupNames()

public static OID getOID(String type)
        throws OIDNotFoundException, OIDNotValidException,
        JaxenException, SAXPathException

public static void setOIDValue(OID oid,  String value)
```

Which is similar to the methods exposed by the facade, however the interface is for objects which carry knowledge of the structure they are querying and manipulating.

The Mibdb object is the working object, which contains the XML structure of the ASN.1 standard for MIBs as XML. It follows the MibManager interface and is the DMS instance specific object, which contains the DMS's state. The Mibdb does nothing more than carry its XML database and query against it. The MibService serves mainly as a facade to the more complicated Mibdb working object.

The *OID* object represents a Mib point and contains the publicly accessible methods;

```
public String getName()

public String getOID()

public String getType()

public String getSystem()

public String getSyntax()

public String getSize()

public String getAccess()

public String getStatus()

public String getDescription()

public String getValue()

public void setValue(boolean value)

public void setValue(int value)

public void setValue(java.lang.String value)

public int getX()

public void setX(int value)

public int getY()

public void setY(int value)

public byte[] getByteValue()

public void setByteValue(byte[] value)
```

Which follows the nodes that the ASN.1 standard contains for a Mib point.

The *Mibdb* is an instance specific to the DMS that is instantiated by the CHART2 system when an operator at a management station begins interaction with a DMS. The *Mibdb* is both readable and writeable, allowing the instance that uses this object to store its Mib values in this *Mibdb* instance. The *Mibdb* carries specific methods that allow it to search its contents for the requested data. It does not know or care what it is carrying beyond the DTD for Mibs. It should only know to store and to serve up the data to the requesting Façade. The usage for NTCIP will be to instantiate this with the read only MibPool table for NTCIP.

Mibdb mibdb = new Mibdb(MibPool.getInstance().getPool());

The *Mibdb* class includes the following methods:

```
public java.util.List getMibNames()

public java.util.List getGroupNames()

public CHART2.Utility.Mib.OID getOID(java.lang.String type)
          throws CHART2.Utility.Mib.OIDNotFoundException, CHART2.Utility.Mib.OIDNotValidException

public void setOIDValue(CHART2.Utility.Mib.OID oid)

public void setOIDValue(CHART2.Utility.Mib.OID oid,
  java.lang.String value)
```

The Mibdb is loaded as an empty XML structure from a server by the *MibPoo*l object. The *MibPool* follows the singleton pattern and exists as a single static instance on the servers JVM. The *MibPool* loads the XML, which represents the Mib points and groups for an NTCIP compliant DMS from persistent storage when the CHART2 system is started. Persistent storage can be either from the file system or from the database. As it is only loaded once, the overhead for loading the initial XML into the system is only felt the first time. When an NTCIP CHART2 DMS is instantiated in the CHART2 system, the DMS requests an empty Mibdb. The Mibdb is cloned from the XML template that the *MibPool* is carrying. The *MibPool* has the public methods;

```
public static MibPool getInstance()

public synchronized Mibdb getPool()
```

Where the getInstance() method is the initial manner in which to instantiate the MibPool on the JVM as the MibPool constructor is private. The synchronized method getPool() is the method to request the initial Mibdb for NTCIP compliant Signs.

The Mib groups are fairly unchanging, and as such, can be represented in an XML Document Type definition, which describes the structure the MibXML has to follow. This is the template for the manner in which the data in the Mibdb must be structured.

```
<!ELEMENT root ANY>
<!ELEMENT name (#PCDATA) EMPTY>
<!ELEMENT definitions (#PCDATA) EMPTY>

<!ELEMENT group (description?,mib*,entry*) EMPTY>
<!ATTLIST group
  name CDATA #REQUIRED
  oid CDATA #REQUIRED
  system CDATA #REQUIRED
>

<!ELEMENT mib (description?) EMPTY>
<!ATTLIST mib
  oid CDATA #REQUIRED
  system CDATA #REQUIRED
  type CDATA #REQUIRED
  syntax CDATA #REQUIRED
  size CDATA #IMPLIED
  access CDATA #REQUIRED
  status CDATA #REQUIRED
>

<!ELEMENT description (#PCDATA) EMPTY>

<!ELEMENT entry (description?,mib*) EMPTY>
```

```
<!ATTLIST entry
  oid CDATA #REQUIRED
  system CDATA #REQUIRED
  type CDATA #REQUIRED
  syntax CDATA #REQUIRED
  size CDATA #IMPLIED
  access CDATA #REQUIRED
  status CDATA #REQUIRED
  index CDATA #REQUIRED
>
```

The DTD only guarantees that the structure be maintained in the flatfile representation of the XML when being loaded. The same DTD can be mapped into a Java object named MibDTD for the Mibdb to take advantage of;

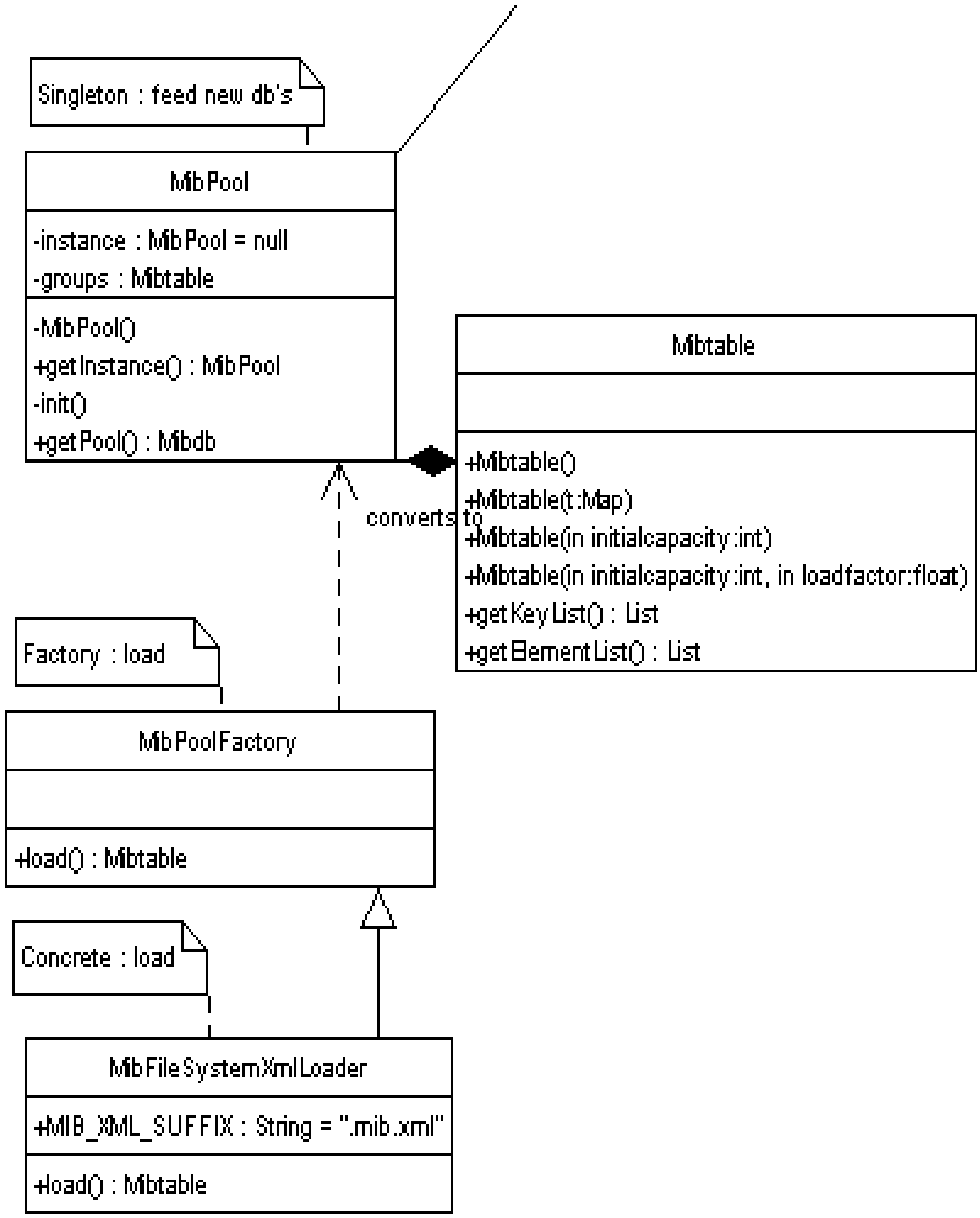
```
/** DTD node value */
public static final String NODE_NAME = "name";

/** DTD node value */
public static final String NODE_DESCRIPTION = "description";

/** DTD attribute value */
 public static final String ATTRIBUTE_OID = "oid";

/** DTD attribute value */
public static final String ATTRIBUTE_TYPE = "type";

/** DTD attribute value */
public static final String ATTRIBUTE_SYSTEM = "system";

/** DTD attribute value */
public static final String ATTRIBUTE_SYNTAX = "syntax";

/** DTD attribute value */
public static final String ATTRIBUTE_SIZE = "size";

/** DTD attribute value */
public static final String ATTRIBUTE_ACCESS = "access";

/** DTD attribute value */
public static final String ATTRIBUTE_STATUS = "status";
```

The Mibtable class is a mib specific object of hash tables for Mibs. The main difference being, returning the keys and elements as lists, rather than enumerations. This is to keep it consistent with other parts of the CHART2.Utility.Mib package. Methods include:

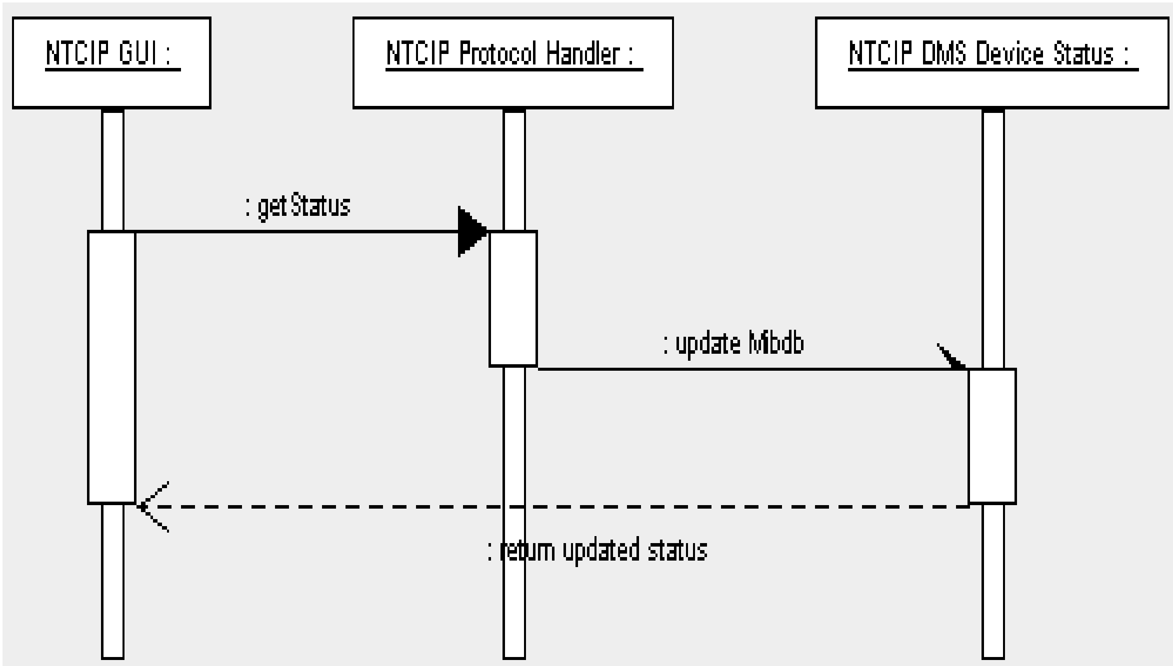
```
/*
 * <p>The Keys from the hash table as the List interface. The keys
 * are returned from the hash table as an enumeration. The List is
 * for package consistency.
 * @return List, a list of the keys in the Mibtable.
 */
public List getKeyList()


/*
 *<p>The Elements form the hash table as the List interface. The elements
 * are returned from the hash table as an enumeration. The List is
 * for package consistency.
 * @return List, a list of the elements in the Mibtable.
 */
public List getElementList()
```

The *MibFileSystemXmlLoader* class extends the factory for loading the XML from its raw format and passing it to the MibPool as a document. The XML loader loads the XML from within the jar. This negates the need for the XML to be in a central repository such as a database or in a client systems file system.

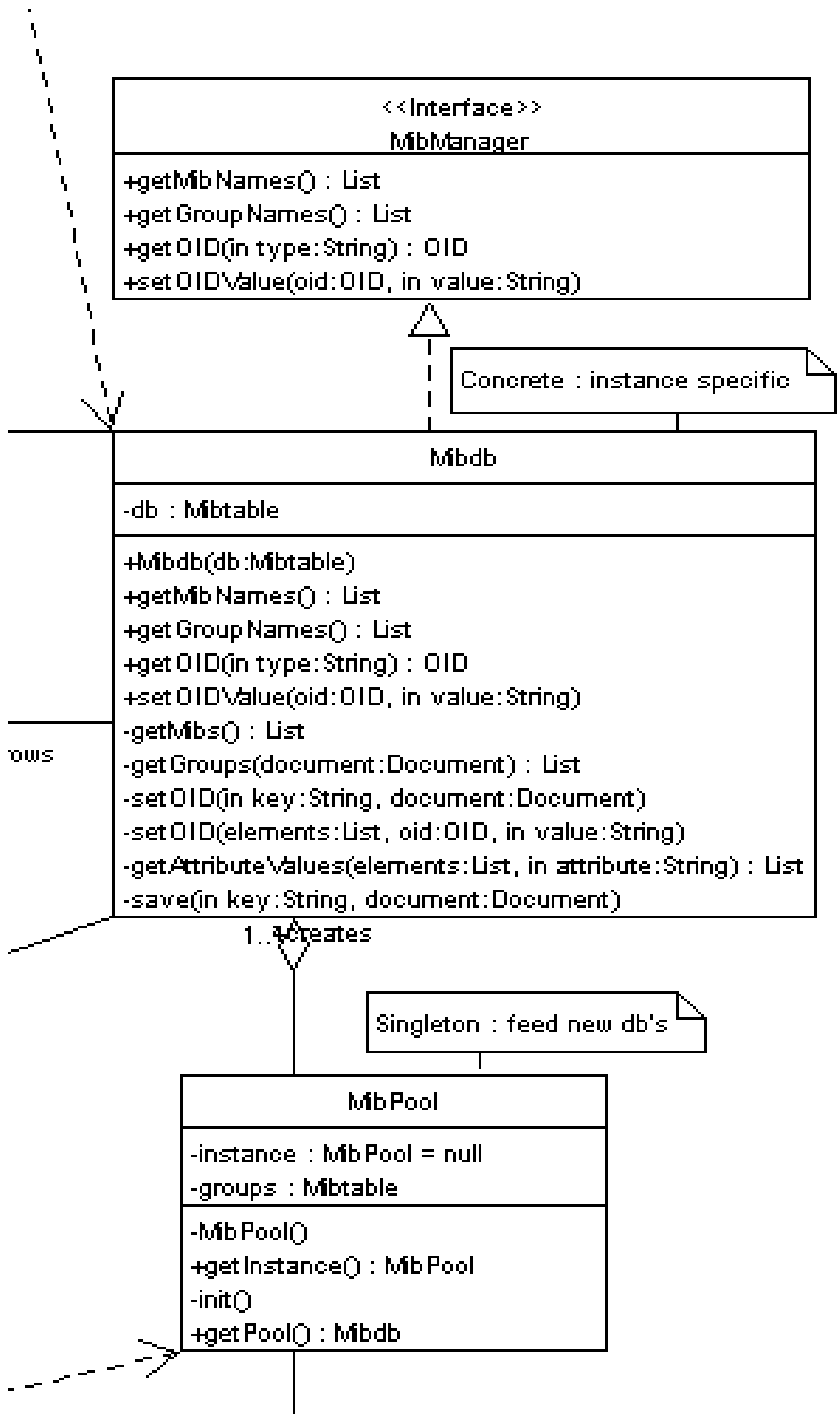
The *MibPoolFactory* class is an abstract factory for the Mibs as XML. Override this to create specific implementations for loading the XML from persistent storage such as file systems, database, serialized objects etc.



UML Diagram for the MibPool, MibTable and the MibPoolFactory.



Sequence Diagram for NTCIP GUI using the Protocol Handler to update the status of the DMS Device.



UML Diagram for the MibManager interface, the Mibdb working object and the Singleton MibPool.

NTCIP DMS Integration into CHART2

The CHART2 system provides a series of strong interfaces, which carry the properties of a DMS. The NTCIP DMS only needs to implement and extend these objects to provide the required CHART2 behavior of the DMS.

A specific DMS in the CHART2 system must adhere to the DMS and CHART2DMS interfaces provided by the CHART2 system. These are discussed in detail in the High Level Design documentation for CHART2. The CHART2DMS interface requires the DMS to carry the CHART2DMSStatus object, which contains information on the current status of the DMS. The CHART2DMSStatus contains the publicly accessible class member variables;

```
m_currentMessage
m_performingPixelTest
m_commMode
m_opStatus
m_shortErrorStatus
m_statusChangeTime
m_controllingOpCenter
```

Where m_currentMessage is a DMSMessage object, m_performingPixelTest is a Boolean value, m_commMode is a CommunicationMode object, m_opStatus is an OperationalManagement object, m_shortErrorStatus is an int, m_statusChangeTime is an int and m_controllingOpCenter is an OpCenterInfo object. These objects predominantly describe the Sign's external status as opposed to the signs Mib state. The DMSMessage is described through MIBs, however the CHART2 DMSMessage object contains more information about the message, such as its Multi-Message format and ASCII representation.

The DMSConfiguration object contains more detail on the DMS's properties such as;

```
m_name
m_deviceLocation
m_dmsSignType
m_signMetrics
m_pages
m_dmsTimeCommLoss
m_dmsBeaconType
m_defaultJustificationLine
m_defaultPageOnTime
m_defaultPageOffTime
```

The CHART2 system also contains in the CHART2.DMSProtocols package a DMSDeviceStatus object, which contains information on the beacon state, the Multi Message and the short Error Status of the DMS. This is the object returned by the Protocol Handler's getStatus() method.

The standard CHART2 DMS GUI interface component will be used by the NTCIP DMS.

The *NTCIPDMSImpl* class object is used for the implementation of the NTCIP DMS. The object implements the IDL generated NTCIPDMSOperations object located in the CHART2.DMSControl package and extends the existing CHART2DMS implementation. The NTCIPDMSOperations object overrides methods in CHART2DMS in order to retrieve NTCIP specific behavior.

Resources

- CHART2 : http://www.chart.state.md.us/
- Edwards and Kelcey Technology : http://www.ekcorp.com/
- NEMA : http://www.nema.org/
- NTCIP : http://www.ntcip.org/
- Object Management Group : http://www.omg.org/

References

- Edwards and Kelcey Report Subtask 1 : NTCIP Compliance Survey and Driver Development. 2001.
- Edwards and Kelcey Report Task 23 : NTCIP implementation of PMPP in CHART2. 2001.
- Edwards and Kelcey Report Task 23 : NTCIP MIBs XML Storage Mechanism. 2001.
- MSHA Report : Performance Evaluation of CHART, An Incident Management Program. 1997.
- MSHA Report : CHARTII Release I, Build 2 High Level Design.
- MSHA Report : CHARTII Release I Build 2 - GUI Detail Design.
- MSHA Report : CHARTII Release I, Build 2 - Field Management Station detailed Design.
- MSHA Report : CHARTII Release I, Build 2 - Field Management Station High Level Design.
- MSHA Report : CHARTII Release I, Build 2A - High Level Design.
- MSHA Report : CHARTII Release I, Build 2A - Detailed Design.

Glossary

- CORBA : Common Object Request Broker Architecture.
- DMS : Dynamic Message Sign.
- DTD : Document Type Definition.
- FMS : Field Management Station.
- GUI : Graphical User Interface.
- HDLC : High-level Data Link Control.
- IDL : Interface Definition Language.
- ISDN : Integrated Services Digital Network.
- MIB : Management Information Base.
- NEMA : National Electrical Manufacturers Association.
- NTCIP : National Transportation Communications for ITS Protocol.
- OID : Object Id.
- ORB : Object Request Broker.
- PDU : Protocol Data Unit.
- PMPP : Point to Multi Point Protocol.
- POA : Portable Object Adapter.
- SNMP : Simple Network Management Protocol.
- VMS : Variable Message Sign.
- WAN : Wide Area Network.
- XML : Extensible Mark-up Language.